REMARKS

The specification has been amended for clarification purposes only, and does not present new matter. Claims 1, 3-6, 8, 10, 11, 13, 15-18, 20, 21, 23, and 25 have been amended. No claims have been canceled or added. Thus, claims 1-25 are currently pending in the case. Further examination and reconsideration of the presently claimed application is hereby respectfully requested.

Interview Summary

In the interview of March 30, 2004, Examiners Cabeca and Bonshock indicated that the amendments proposed to claim 1 would overcome the rejections cited in the Office Action about that claim. In particular, the Examiners agreed that the cited art failed to provide teaching or suggestion for a proxy component which, after being selected by a peer component, dynamically creates a new graphics resource component for displaying an object. Before concluding the interview, Examiners Cabeca and Bonshock expressed concern about the methods recited in claims 13 and 25. Though specific amendments were not discussed, the Examiners suggested that claims 13 and 25 may also be allowable over the cited art if those claims were amended: (i) in a manner similar to the amendments made to independent claim 1, and (ii) to clarify the structure performing the claimed methodologies. As noted above, the software components of claims 13 and 25 have been amended to, instead, be proxy components which could be activated (one at a time) to dynamically generate the graphical representations during runtime. Therefore, claims 13 and 25 have been amended in a manner similar to the amendments discussed for claim 1. The method of claim 13 was further amended to specify the structure (i.e., the application program) responsible for performing the method steps recited in present claims 13 and 25 (note: independent claim 25 included this structure in it's pre-amendment state, and therefore, required no further amendment).

The amendments provided herein are believed to clarify the claim language in a manner that addresses the concerns expressed in the Office Action, as well as those concerns expressed by Examiners Cabeca and Bonshock in the interview of March 30, 2004. In light of these amendments, Applicants respectfully request that the current rejections be removed. Arguments supporting the patentability of the presently claimed case over the cited art are provided in more detail below.

Objection to the Specification

The Specification was objected for an informality. In particular, the Office Action suggests that on page 41, the statement in the paragraph labeled by the number 2 is in the form of an if-then statement where the then or assumed then is missing. To expedite prosecution, the Specification has been amended in a manner that addresses those concerns expressed in the Office Action. Accordingly, removal of this objection is respectfully requested.

Section 103 Rejections

Claims 1-25 were rejected under 35 U.S.C. § 103(a) as being unpatentable over WinZip computing Inc., WINZIP 8.0 (hereinafter "WinZip") in view of the publication referred to as Java Platform 1.2 Beta 4 API Specification: Class JPasswordField and Class JTextField (hereinafter "Java"). To establish a *prima facie* obviousness of a claimed invention, all claim limitations must be taught or suggested by the prior art. *In re Royka*, 490 F.2d 981, 180 U.S.P.Q. 580 (C.C.P.A. 1974), MPEP 2143.03. Obviousness cannot be established by combining or modifying the teachings of the prior art to produce the claimed invention, absent some teaching or suggestion or incentive to do so. *In re Bond*, 910 F. 2d 81, 834, 15 USPQ2d 1566, 1568 (Fed. Cir. 1990). The cited art does not teach or suggest all limitations of the currently pending claims, some distinctive limitations of which are set forth in more detail below.

None of the cited art teaches or suggests a system of software components that includes a first proxy component, a second proxy component and a peer component for displaying an object, where the peer component is configured for selecting, during runtime, either the first proxy component or the second proxy component, depending on a mode of use of the object, and where the selected proxy component dynamically creates a new graphics resource component for displaying the object, such that the appearance of the object is substantially independent of an operating system. Amended independent claim 1 states in part:

> A system of software components adapted to display an object... wherein the system of software components comprises: a first proxy component; a second proxy component; and a peer component for selecting either the first proxy component or the second proxy component, depending on a mode of use of the object, wherein the selection can be made during runtime, and wherein the selected proxy component thereafter creates a new graphics resource component for displaying the object, such that the appearance of the displayed object is substantially independent of the operating system.

Support for the amendment to claim 1 may be found in the Specification, for example, on page 38, line 17 to page 40, line 6; and *see* Figs. 17 and 18. Support for the dynamic creation of a new graphics resource component may be found on page 39, lines 7-25 of the Specification.

The Specification provides a unique system of software components that enables an application program (e.g., a Java application) to be truly portable across all operating system (OS) platforms by providing various means for maintaining the "look and feel" of the application program. "A software program is said to be 'portable' across various platforms if the program can run without modification on any of those platforms." (Specification, page 6, lines 23-24). Before disclosing such means, the Specification highlights several drawbacks of prior art attempts at application portability.

For example, Java application programs utilize a platform-dependent application program interface (API), commonly known as the Abstract Windowing Toolkit (AWT), to produce heavyweight software components that are written in native code (i.e., instructions specific to a particular OS). When AWT heavyweight software components are used for displaying images, the "look and feel" of those images may differ depending on the particular OS running the Java application program. In the context of a graphical user interface (GUI), "the 'look and feel' of a GUI refers to such things as the appearance, color and behavior of Buttons, TextFields, Listboxes, menus, etc..." (Specification, page 7, line 4-8). Thus, there are certain limitations within the AWT that prohibit true portability of Java application programs, especially in terms of the look and feel of the application program. *See*, e.g., Specification, page 7, line 24 to page 9, line 10, and page 18, line 10 to page 19, line 23.

In an effort to overcome the platform-dependency of Java application programs, Swing was developed as part of the Java Foundation Classes (JFC). An API written using Swing contains no native code, and therefore, can be run on substantially any OS without changing the look and feel of the application. *See*, e.g., Specification, page 9, lines 14-30, and page 19, line 25 to page 20, line 4. Unfortunately, the lightweight software components of Swing cannot completely eliminate the platform-dependency of Java applications that use AWT. Since Swing versions of many AWT components (e.g., containers, such as frames) are unavailable, many programmers have attempted to mix Swing and AWT components within a given API. *See*, e.g., Specification, page 21, lines 1-13. However, straightforward mixing of Swing and AWT components tends to create many problems that programmers have simply learned to accept. *See*, e.g., Specification, page 22, lines 1-28.

Therefore, the presently claimed case provides a system, method and computer-readable storage device for overcoming one of the many problems that occurs when programmers attempt to mix AWT and Swing components within a Java application program. In general, the presently claimed case provides a system of software components – referred to as "AWTSwing" components -- that enable an object to be displayed using one of two different Swing software components, which may be dynamically created at runtime depending on a mode of use of the object. In other words, the presently claimed system of software components provides a unique mode-switching capability that allows two Swing software components to alternate as replacements for an AWT software component, depending on the manner in which the object is being used. *See*, e.g., Specification, page 38, line 17 to page 40, line 6.

In one embodiment, the presently claimed system of software components may allow the Swing JTextField and JPasswordField software components to alternate as replacements for the AWT TextField software component. In AWT, the functionalities of a normal text field and a password-protected text field are both included within the AWT TextField component. However, the normal and password-protected functionalities are not included within a single Swing component, but instead, are separately allocated to the Swing JTextField and JPasswordField components. If a Java application program originally contains an AWT TextField component, the behavior of that program cannot be maintained by simply replacing the AWT TextField component with a single Swing component. In order to utilize Swing components, while maintaining behavioral compatibility with the Java application program, the appropriate Swing component must be dynamically recreated and substituted for the AWT component each time the mode of use of the object changes. This is in direct contrast to the manner in which Swing components are normally created (i.e., during the initial construction of the peer component). *See*, e.g., Specification, page 39, lines 7-8.

To provide mode-switching capability, the presently claimed system of software components includes a first proxy component (e.g., JTextFieldProxy 174), a second proxy component (e.g., JPasswordFieldProxy 176) and a peer component (e.g., JTextFieldPeer 172), as shown in FIG. 17. As used herein, the term "software component" may be defined as "any sequence of executable code." (Specification, page 3, line 16). As such, a "proxy component" may be generally defined as a sequence of executable code that, when executed, (i) translates the method calls from an AWT component to the appropriate Swing component(s), and (ii) translates the event calls from a Swing component to the AWT component (e.g., a frame) containing the Swing component. See, e.g., Specification, page 29, lines 7-10. As used by the proxy component, a "method call" may be defined as a get routine to another software

component (i.e., a "graphics resource component") that creates and manipulates a graphical representation of the object to therefore display the object. *See*, e.g., Specification, page 23, lines 16-18.

In present claim 1, the "peer component" is used for selecting either the first proxy component or the second proxy component, depending on a mode of use of the object. Once selected by the peer component, the selected proxy component will create a new graphics resource component (e.g., a Swing JTextField or JPasswordField software component) for displaying the object. Since the object is displayed by a Swing component, rather than an AWT component, the appearance of the displayed object will be substantially independent of the operating system running the Java application program. In other words, the properties that define the "look and feel" of the object (e.g., the color, position, text, etc.) will not be affected by the operating system running the Java application program.

Neither WinZip nor Java, either separately or in combination, can be used to teach or suggest the presently claimed system of software components. For example, the screen shots provided on pages 3 and 4 of the WinZip reference can only be used to show how various objects (e.g., frames, dialog boxes, buttons, check boxes, text boxes, etc.) may be displayed when the "Mask password" checkbox is toggled between unchecked (page 3) and checked (page 4). More specifically, the screen shots on pages 3 and 4 of the WinZip reference can only be used to show how the Password text box may display masked text ("*****") or unmasked text ("hello"), depending on a mode of use of the Password text box.

However, the screen shots on pages 3 and 4 of the WinZip reference cannot be used to provide teaching or suggestion for the presently claimed system of software components, which as noted above, include a first proxy component, a second proxy component and a peer component. Though the screen shots may show the end product (i.e., the WinZip GUI) obtained by the execution of one or more (inherently included) WinZip software components, the screen shots cannot provide insight into the particular software components (where "software components" are defined as sequences of executable code) that were used to produce the end product. The WinZip reference simply does not and cannot teach or suggest that the presently claimed software components -- i.e., the first proxy component, the second proxy component and the peer component – are used for displaying the objects shown in the WinZip GUI.

The WinZip reference also fails to provide inherent teaching or suggestion for the presently claimed system of software components. For example, the WinZip reference does not teach or suggest that the functionality shown in the screen shots (i.e., the alternation between masked and unmasked text in the

Password text box) could be provided by software components similar to the presently claimed proxy and peer components. In other words, given the functionality disclosed in the screen shots on pages 3 and 4 of the WinZip reference, one skilled in the art could not reasonably conclude that a first proxy component, a second proxy component and a peer component (or equivalent software components demonstrating the presently claimed functionality) would necessarily be included to provide such functionality.

First of all, the WinZip reference fails to disclose the programming language used to produce the WinZip GUI. The Applicant concedes that one skilled in the art may assume that an object-oriented programming (OOP) language was used to construct the software components needed to produce the WinZip GUI, simply because such languages are typically used in developing GUIs. Thus, for the sake of argument, we may assume that the WinZip screen shots are produced by one or more software components written purely in the Java programming language (a popular OOP language, and therefore, a reasonable assumption). After making such an assumption, one skilled in the art may also assume that a Java peer component (e.g., the heavyweight AWT TextComponentPeer) and a Java graphics resource component (e.g., the heavyweight AWT TextField) are included within the WinZip software components. If this is the case, the Java peer component could invoke the methods of the Java graphics resource component, which would then display the object. However, the appearance of the displayed object would not be independent of the operating system, as in the presently claimed case, because a heavyweight graphics resource component (containing native code) would be used for displaying the object in the WinZip GUI. Furthermore, since the methods for displaying text as either masked or unmasked are both provided within the existing Java graphics resource component (e.g., within the AWT TextField component), the Java peer component would have no need for selecting between a first proxy component and a second proxy component. In fact, if the WinZip software components were written purely in Java, there would be no need for including proxy components that, when selected by the peer component, function to dynamically create new graphics resource components for displaying the object. Thus, if one were to assume that the WinZip software components were written purely in Java, the WinZip software components would not read upon the presently claimed system of software components.

The WinZip reference provides absolutely no teaching or suggestion for combining AWT and Swing software components within a Java application program that, when executed, displays an object in a manner that is independent of the operating system running the Java application. However, on page 3 of the Office Action, the Examiner suggests that it would have been obvious to one skilled in the art "to modify the text display system of WinZip to include the system independence of [the] Java [reference – not

the language]." The Applicant respectfully disagrees, for at least the reasons set forth in more detail below.

The Java reference is merely an overview, or a brief description, of the Swing JTextField and JPasswordField components. Like the WinZip reference, the Java reference fails to provide any teaching or suggestion for combining AWT and Swing software components within a Java application program. Neither reference indicates that AWT and Swing software components can be combined in a manner that provides the presently claimed system of software components or the functionality thereof. Thus, the cited art fails to provide motivation for modifying the text display of the WinZip reference (which could reasonably include AWT components) to include the system independence of the Java references (which could be provided by Swing components). The Applicant asserts that the teaching or suggestion to make the claimed combination and the reasonable expectation of success must both be found in the prior art, not in the Applicant's disclosure. *In re Vaech*, 947 F.2d 488, 20 USPQ2d 1438 (Fed. Cir. 1991); emphasis added, MPEP 2143.

If one were to make the combination proposed by the Examiner (even without sufficient motivation to do so), the proposed combination would still fail to disclose all limitations of present claim 1. Lets assume, for example, that a programmer wishes to replace the above-mentioned Java graphics resource component (e.g., the heavyweight AWT TextField) with a Swing graphics resource component, i.e., a lightweight software component. If the programmer attempted to make such a replacement, he/she would run into several problems.

First of all, the Java graphics resource component may have combined features (e.g., masked and unmasked text capabilities) that simply do not exist within a single Swing graphics resource component. Therefore, the programmer could not directly replace the Java component with an equivalent Swing component (since none exists). Even if the programmer were to replace the Java graphics resource component with multiple Swing graphics resource components (e.g., by modifying the existing Java peer component to invoke the methods of the Swing, rather than the AWT, graphics resource components), the replacement Swing components would not behave properly (e.g., they would not respond to action events, such as mouse clicks or keyboard entries), since the replacement Swing components would not be declared as contained within a Frame of the Java application program. *See*, e.g., Specification, page 22, lines 23-28.

In addition, one skilled in the art would not consider the presently claimed proxy components to be an obvious feature in light of the WinZip and Java references. For example, Swing components are normally created only during the initial construction of the peer component (*See*, e.g., Specification page 39, lines 7-25). Therefore, one skilled in the art would not consider the dynamic creation of Swing components to be an obvious feature. As a consequence, the presently claimed proxy components (which dynamically create the appropriate Swing component during runtime) cannot be considered obvious features in light of the WinZip and Java references.

None of the cited art teaches or suggests an application program for activating a first proxy component to dynamically generate a first graphical representation of an object, and upon detecting a change in the mode of use of the object, deactivating the first proxy component and then activating a second proxy component to dynamically generate a second graphical representation of an object, where the first and second graphical representations (one being distinct from the other) are substantially independent of an operating system. Amended independent claim 25 recites in part:

> A computer-readable storage device, comprising ... an application program running under the operating system ... wherein the application program is adapted for: activating a first proxy component to dynamically generate a first graphical representation of the object during runtime that is substantially independent of the operating system; monitoring the mode of use of the object; and upon detecting a change in the mode of use of the object, deactivating the first proxy component and activating a second proxy component to dynamically generate a second graphical representation of the object during runtime, wherein the second graphical representation is substantially independent of the operating system and distinct from the first graphical representation.

Amended independent claim 13 recites a similar limitation. Support for the amendment to claims 13 and 25 may be found in the Specification, for example, on page 38, line 17 to page 40, line 6, and more specifically, on page 39, lines 7-25 of the Specification. Independent claims 13 and 25 recite similar limitations to present claim 1 in that they disclose various method steps that may be performed upon execution of the system of software components recited in present claim 1.

On page 7 of the Office Action, the Examiner suggests that the WinZip reference fails to provide any teaching or suggestion for the appearance of the displayed object (i.e., the graphical representation of the object) being independent of the operating system. The Applicant agrees. However, the Applicant does not agree that the teachings of the WinZip and Java references can be combined to render the above limitation obvious, as is also suggested by the Examiner. As noted in the previous argument, neither

reference provides sufficient motivation that would enable one skilled in the art to make the proposed combination. The Examiner cannot rely on the Applicant's disclosure for providing such motivation.

For at least the reasons set forth above, none of the cited art, either separately or in combination provides motivation to teach or suggest all limitations of present claims 1, 13, and 25. Therefore, claims 1, 13, and 25, as well as claims dependent therefrom, are patentably distinct over the cited art. Accordingly, removal of the § 103(a) rejections of claims 1-25 is respectfully requested.

CONCLUSION

This response constitutes a complete response to all issues raised in the Office Action mailed January 15, 2004. In view of the remarks traversing rejections, Applicants assert that pending claims 1-25 are in condition for allowance. If the Examiner has any questions, comments, or suggestions, the undersigned attorney earnestly requests a telephone conference.

No fees are required for filing this amendment; however, the Commissioner is authorized to charge any additional fees which may be required, or credit any overpayment, to Conley Rose, P.C. Deposit Account No. 03-2769/5468-07600.

Respectfully submitted,

Kevin L. Daffer
Reg. No. 34,146
Attorney for Applicant(s)

Conley Rose, P.C.
P.O. Box 684908
Austin, TX 78768-4908
Ph: (512) 476-1400
Date: April 15, 2004
JMF